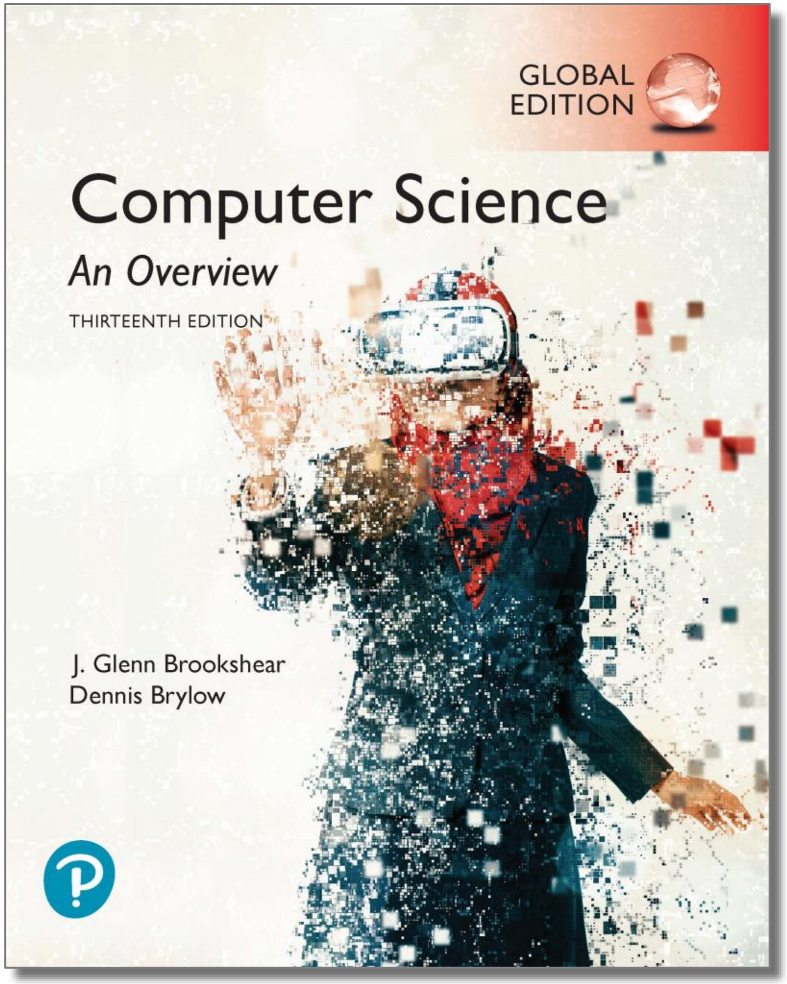
Computer Science An Overview

13th Edition, Global Edition



Chapter 5

Algorithms

Chapter 5: Algorithms

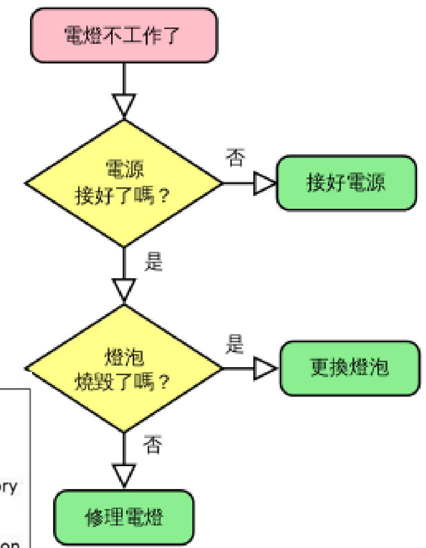
- 5.1 The Concept of an Algorithm
- 5.2 Algorithm Representation
- 5.3 Algorithm Discovery
- 5.4 Iterative Structures
- 5.5 Recursive Structures
- 5.6 Efficiency and Correctness

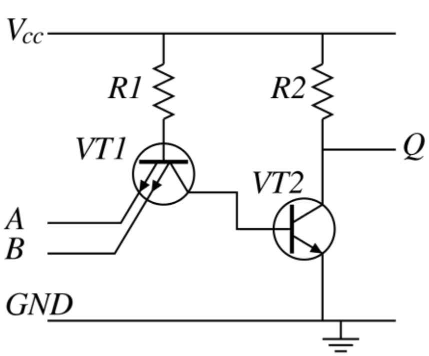
5.1 The Concept of an Algorithm

- Algorithms from previous chapters
  - Converting from one base to another
  - Correcting errors in data
  - Compression
- Many researchers believe that every activity of the human mind is the result of an algorithm

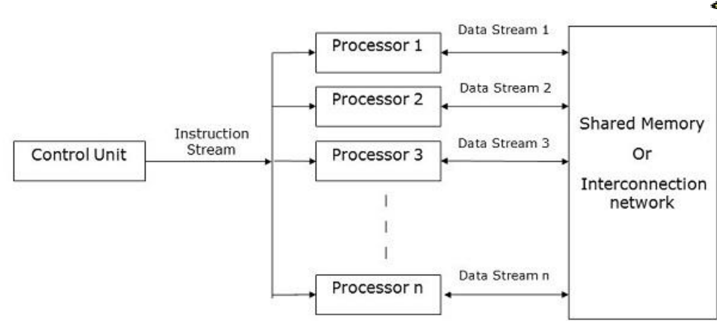
Formal Definition of Algorithm

- An algorithm is an ordered set of unambiguous, executable steps that defines a terminating process
- The steps of an algorithm can be sequenced in different ways
  - Linear (1, 2, 3, …)
  - Parallel (multiple processors)
  - Cause and Effect (circuits)





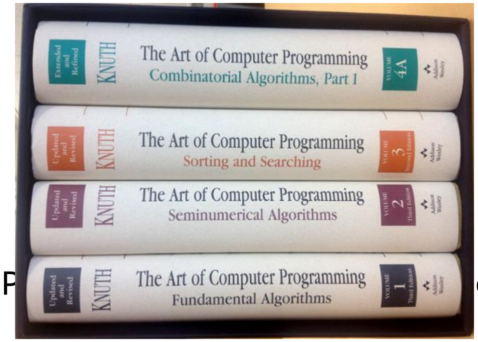
Truth Table		
Input A	Input B	Output Q
0	0	1
0	1	1
1	0	1
1	1	0



Formal Definition of Algorithm

高德納 (Donald Ervin Knut) 在他的著作《電腦程式設計藝術》裡對演算法的特徵歸納：

1. 輸入：一個演算法必須有零個或以上輸入量。
2. 輸出：一個演算法應有一個或以上輸出量，輸出量是演算法計算的結果。
3. 明確性：演算法的描述必須無歧義，以保證演算法的實際執行結果是精確地符合要求或期望，通常要求實際執行結果是確定的。
4. 有限性：依據圖靈的定義，一個演算法是能夠被任何圖靈完備系統類比的一串運算，而圖靈機只有有限個狀態、有限個輸入符號和有限個轉移函式（指令）。而一些定義更規定演算法必須在有限個步驟內完成任務。
5. 有效性：又稱可行性。能夠實現，演算法中描述的操作都是可以通過已經實現的基本運算執行有限次來實現。



The Abstract Nature of Algorithms

- There is a difference between an algorithm and its representation.
  - Analogy: difference between a story and a book
- A Program is a representation of an algorithm.
- A Process is the activity of executing an algorithm.

Formal Definition of Algorithm

- A Terminating Process
  - Culminates with a result
  - Can include systems that run continuously
    - Hospital systems
    - Long Division Algorithm
- A Non-terminating Process
  - Does not produce an answer
  - Chapter 12 : “Non-deterministic Algorithms”

5.2 Algorithm Representation

- Is done formally with well-defined Primitives
  - A collection of primitives constitutes a programming language.
- Is done informally with Pseudocode
  - Pseudocode is between natural language and a programming language.

Figure 5.2 Folding a bird from a square piece of paper

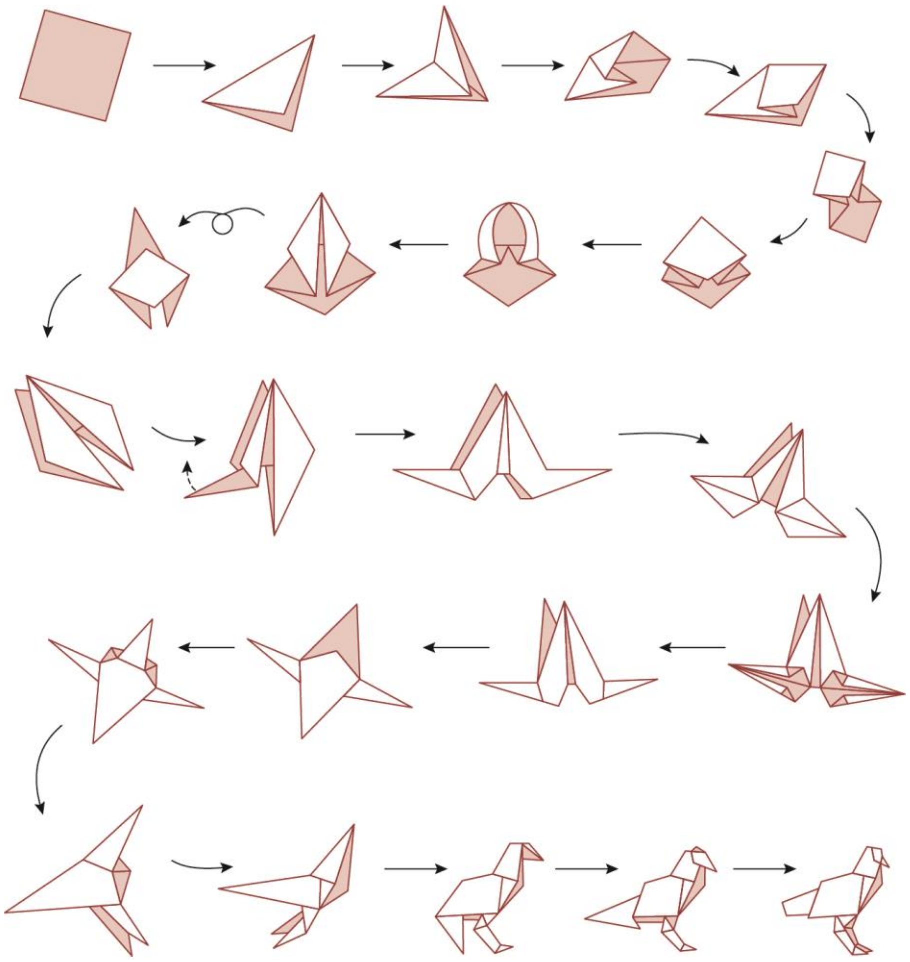
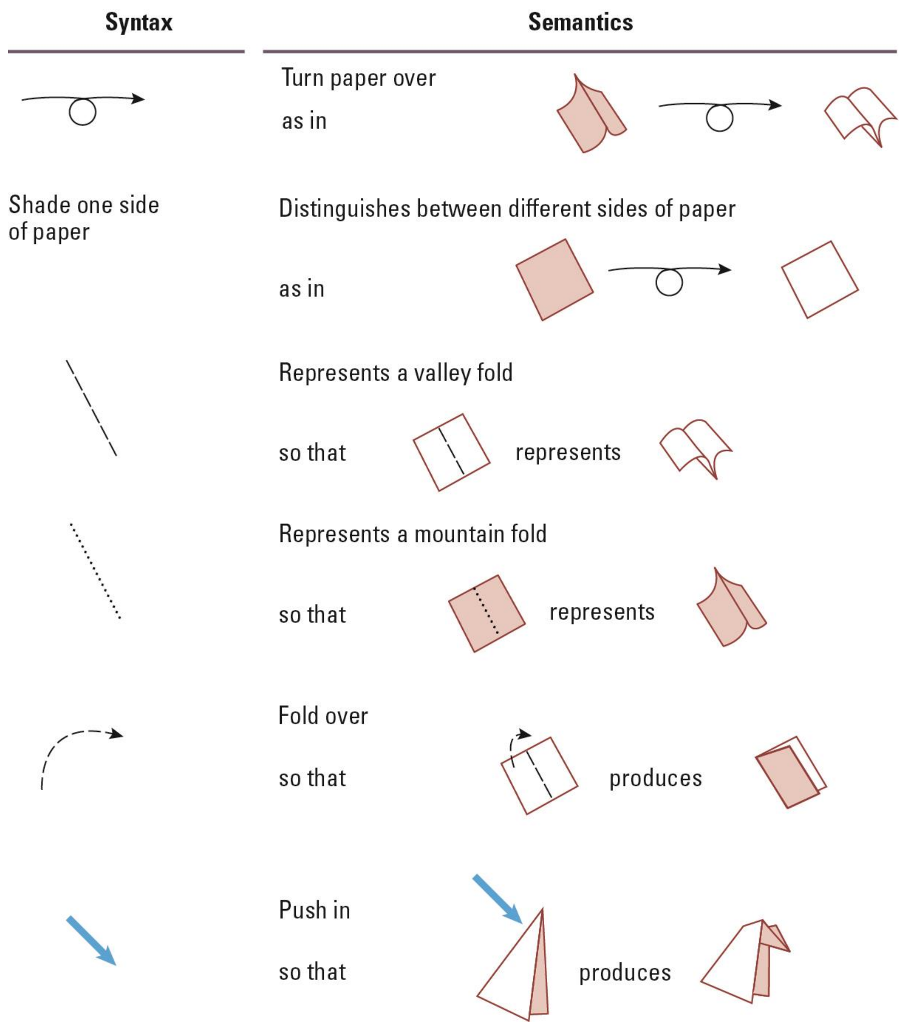


Figure 5.3 Origami primitives



Designing a Pseudocode Language

- Choose a common programming language
- Loosen some of the syntax rules
- Allow for some natural language
- Use consistent, concise notation
- We will use a Python-like Pseudocode

Pseudocode Primitives

- Assignment

    name = *expression*

- example

```
RemainingFunds = CheckingBalance +
                          SavingsBalance
```

Pseudocode Primitives (continued)

- Conditional selection

```
if (condition):
    activity
```

- example

```
if (sales have decreased):
    lower the price by 5%
```

Pseudocode Primitives (continued)

- Conditional selection

```
if (condition):
    activity
else:
    activity
```

- example

```
if (year is leap year):
    daily total = total / 366
else:
    daily total = total / 365
```

Pseudocode Primitives (continued)

- Repeated execution

```
while (condition):
    body
```

- example

```
while (tickets remain to be sold):
    sell a ticket
```

Pearson

Pseudocode Primitives (continued)

- Indentation shows **nested** conditions

```
if (not raining):
    if (temperature == hot):
        go swimming
    else:
        play golf
else:
    watch television
```

Pseudocode Primitives (continued)

- Define a function

```
def name():
```

- example

```
def ProcessLoan():
```

- Executing a function

```
if (. . .):
    ProcessLoan()
else:
    RejectApplication()
```

Figure 5.4 The procedure Greetings in pseudocode

```
def Greetings():
    Count = 3
    while (Count > 0):
        print('Hello')
        Count = Count - 1
```

Pseudocode Primitives (continued)

- Using parameters

```
def Sort(List):
      .
      .
```

- Executing Sort on different lists

```
Sort(the membership list)
Sort(the wedding guest list)
```

5.3 Algorithm Discovery

- The first step in developing a program
- More of an art than a skill
- A challenging task

Polya's Problem Solving Steps

- 1. Understand the problem.
- 2. Devise a plan for solving the problem.
- 3. Carry out the plan.
- 4. Evaluate the solution for accuracy and its potential as a tool for solving other problems.

Polya's Steps in the Context of Program Development

- 1. Understand the problem.
- 2. Get an idea of how an algorithmic function might solve the problem.
- 3. Formulate the algorithm and represent it as a program.
- 4. Evaluate the solution for accuracy and its potential as a tool for solving other problems.

Getting a Foot in the Door

- Try working the problem backwards
- Solve an easier related problem
  - Relax some of the problem constraints
  - Solve pieces of the problem first (bottom up methodology)
- Stepwise refinement: Divide the problem into smaller problems (top-down methodology)

Ages of Children Problem

- Person A is charged with the task of determining the ages of B’s three children.
  - B tells A that the product of the children’s ages is 36.
  - A replies that another clue is required.
  - B tells A the sum of the children’s ages.
  - A replies that another clue is needed.
  - B tells A that the oldest child plays the piano.
  - A tells B the ages of the three children.
- How old are the three children?

Figure 5.5

a. Triples whose product is 36

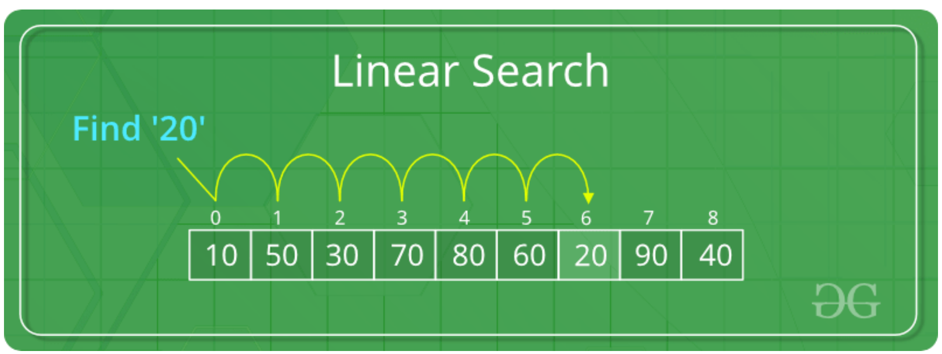
(1,1,36)	(1,6,6)
(1,2,18)	(2,2,9)
(1,3,12)	(2,3,6)
(1,4,9)	(3,3,4)

b. Sums of triples from part (a)

1 + 1 + 36 = 38	1 + 6 + 6 = 13
1 + 2 + 18 = 21	2 + 2 + 9 = 13
1 + 3 + 12 = 16	2 + 3 + 6 = 11
1 + 4 + 9 = 14	3 + 3 + 4 = 10

5.4 Iterative Structures

- A collection of instructions repeated in a looping manner
- Examples include:
  - Sequential Search Algorithm
  - Insertion Sort Algorithm



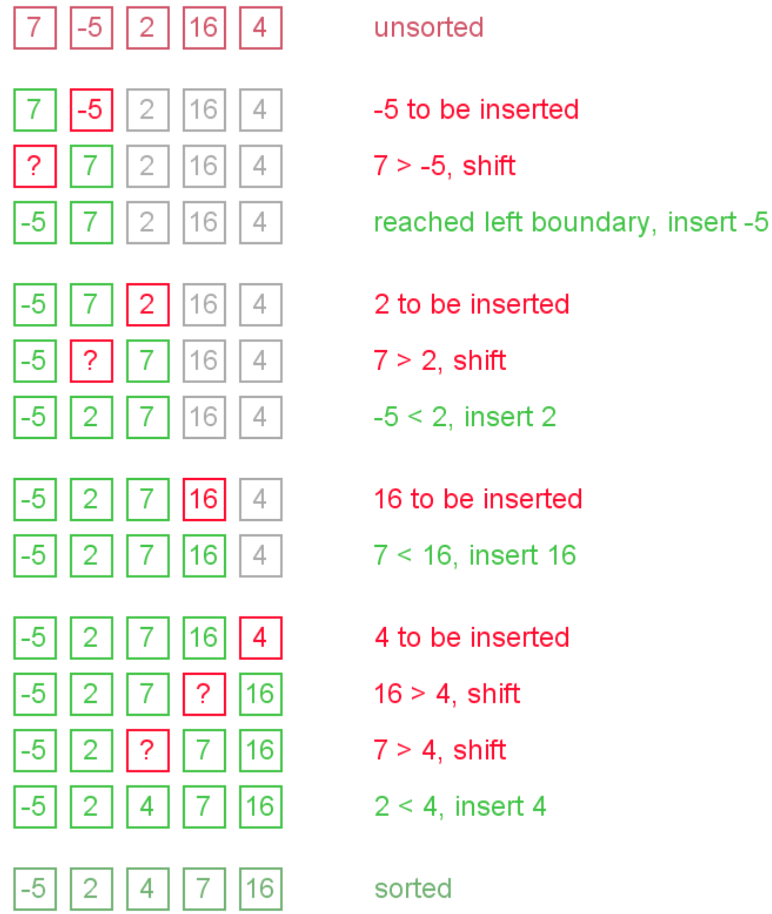


Figure 5.6 The sequential search algorithm in pseudocode

```
def Search (List, TargetValue):
    if (List is empty):
       Declare search a failure
    else:
       Select the first entry in List to be TestEntry
       while (TargetValue != TestEntry and entries remain):
            Select the next entry in List as TestEntry
       if (TargetValue == TestEntry):
            Declare search a success
       else:
            Declare search a failure
```

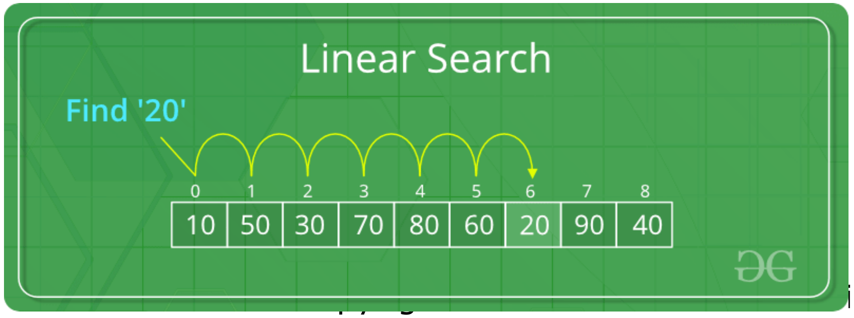


Figure 5.7 Components of repetitive control

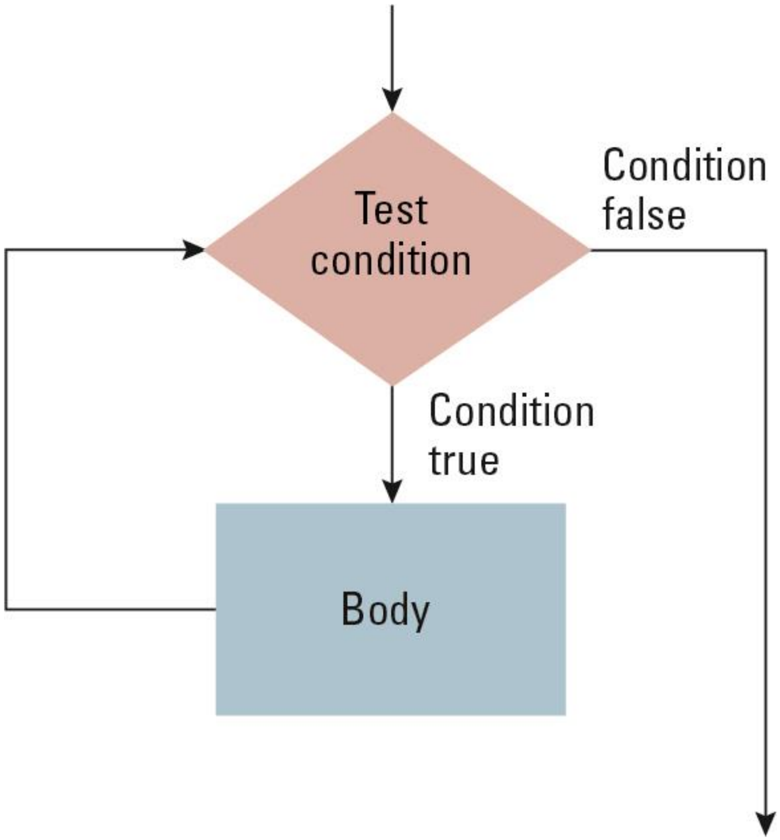
Initialize:	Establish an initial state that will be modified toward the termination condition
Test:	Compare the current state to the termination condition and terminate the repetition if equal
Modify:	Change the state in such a way that it moves toward the termination condition

Iterative Structures

- Pretest loop:

```
while (condition):
  body
```

The while loop structure



Iterative Structures

- Posttest loop:

```
repeat:
    body
    until(condition)
```

The repeat loop structure

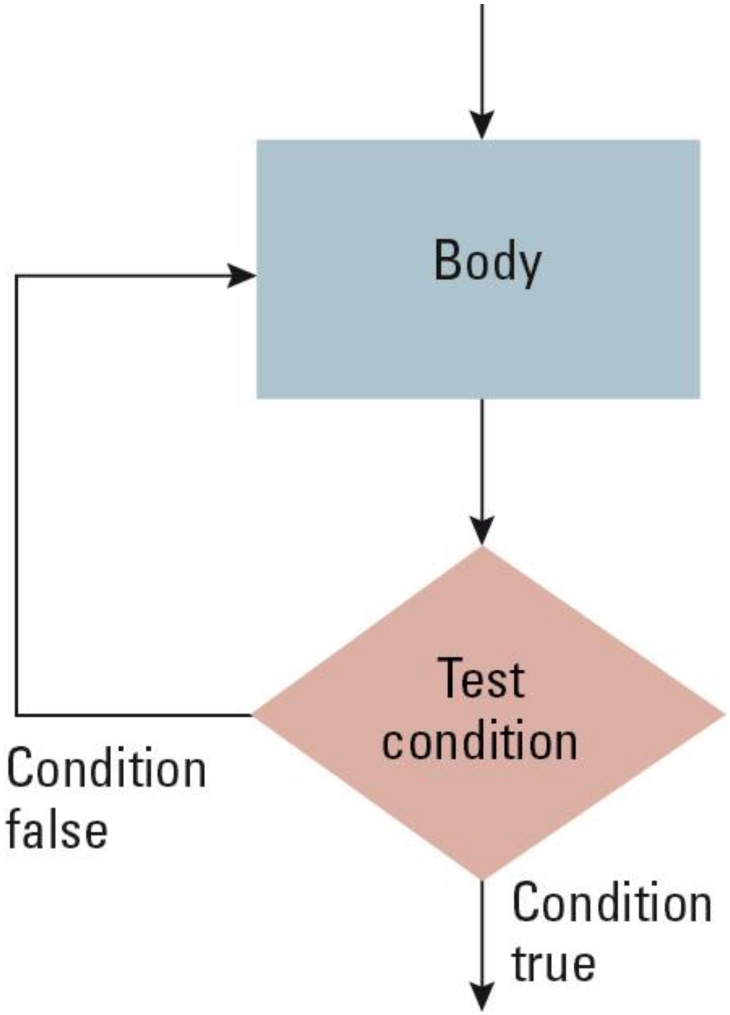


Figure 5.10 Sorting the list Fred, Alex, Diana, Byron, and Carol alphabetically

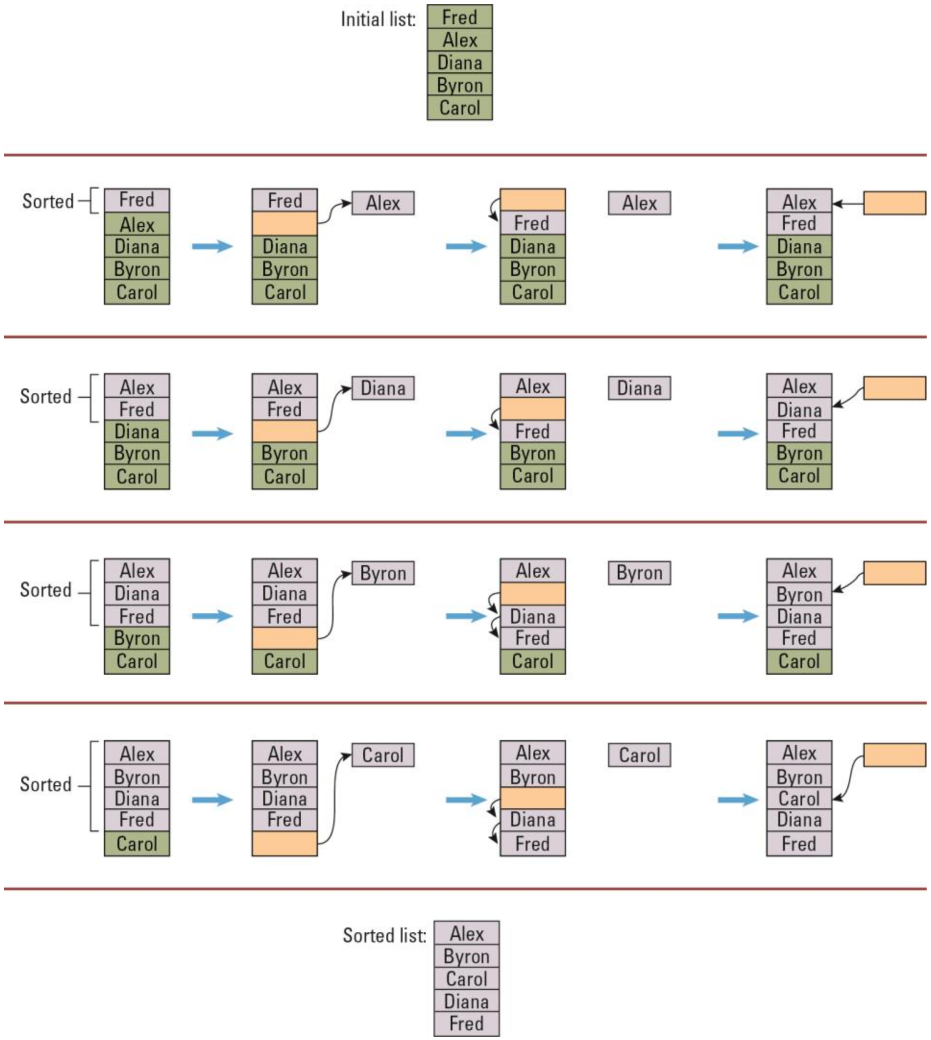


Figure 5.11 The insertion sort algorithm expressed in pseudocode

```
def Sort(List):
    N = 2
    while (N <= length of List):
        Pivot = Nth entry in List
        Remove Nth entry leaving a hole in List
        while (there is an Entry above the
                    hole and Entry > Pivot):
            Move Entry down into the hole leaving
            a hole in the list above the Entry
        Move Pivot into the hole
        N = N + 1
```

Pearson

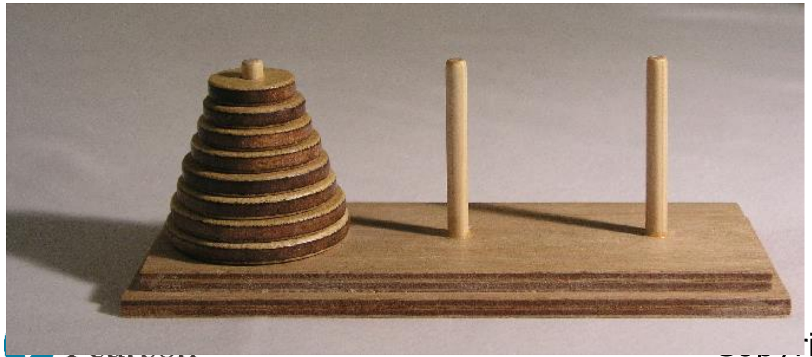
5.5 Recursive Structures

- Repeating the set of instructions as a subtask of itself.
- Multiple activations of the procedure are formed, all but one of which are waiting for other activations to complete.
- Example: Tower of Hanoi, The Binary Search Algorithm

5.5 Recursive Structures

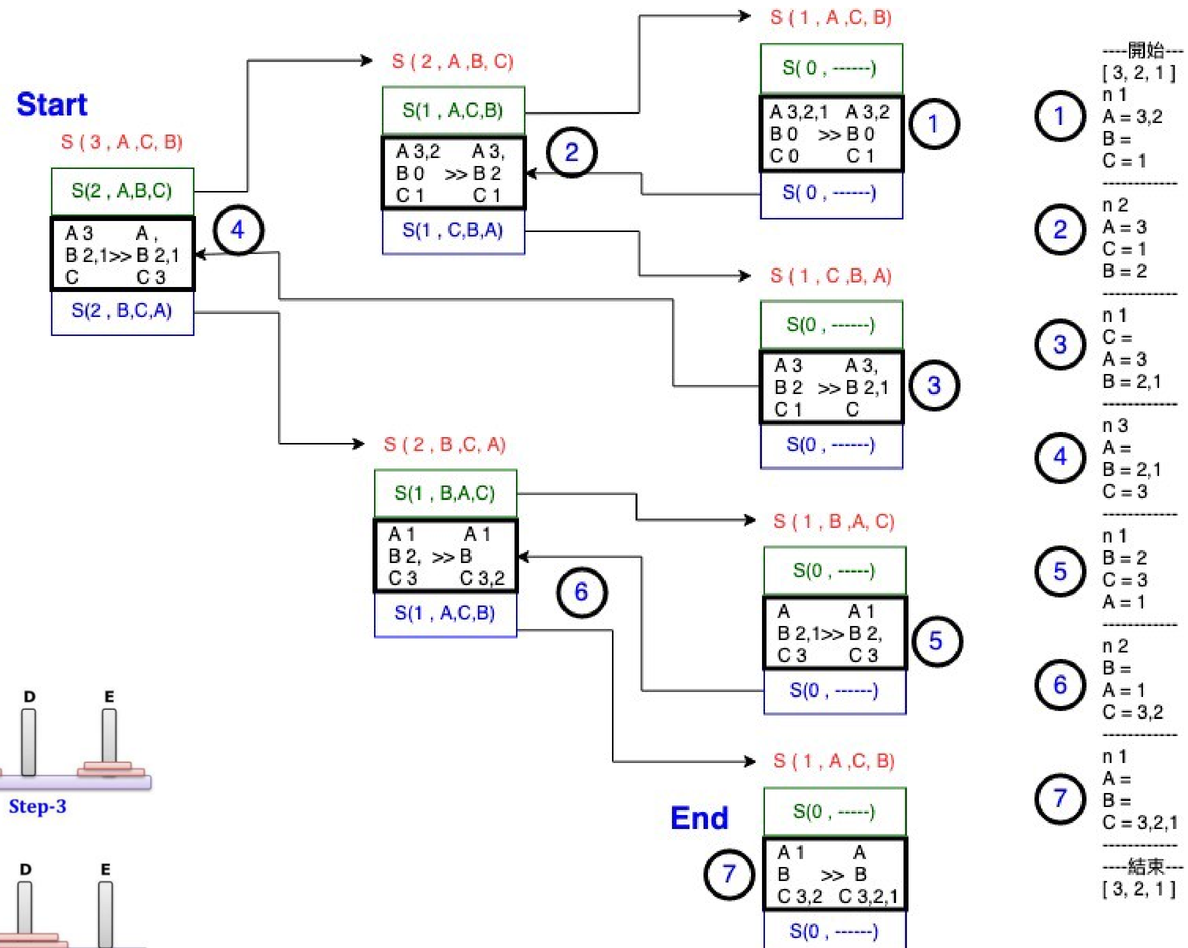
- Tower of Hanoi

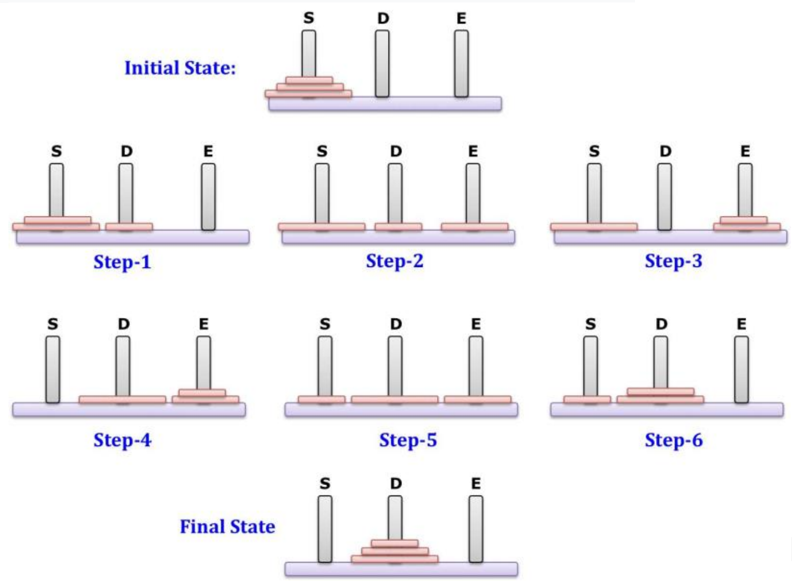
1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a disk that is smaller than it.



5.5 Recursive Structures

```python
def hanoi(n, a, b, c):
    if n == 1:
        print(a, '-->', c)
    else:
        hanoi(n - 1, a, c, b)
        hanoi(1    , a, b, c)
        hanoi(n - 1, b, a, c)
# 调用
if __name__ == '__main__':
    hanoi(5, 'A', 'B', 'C')
```





5.5 Recursive Structures

- Example: The Binary Search Algorithm

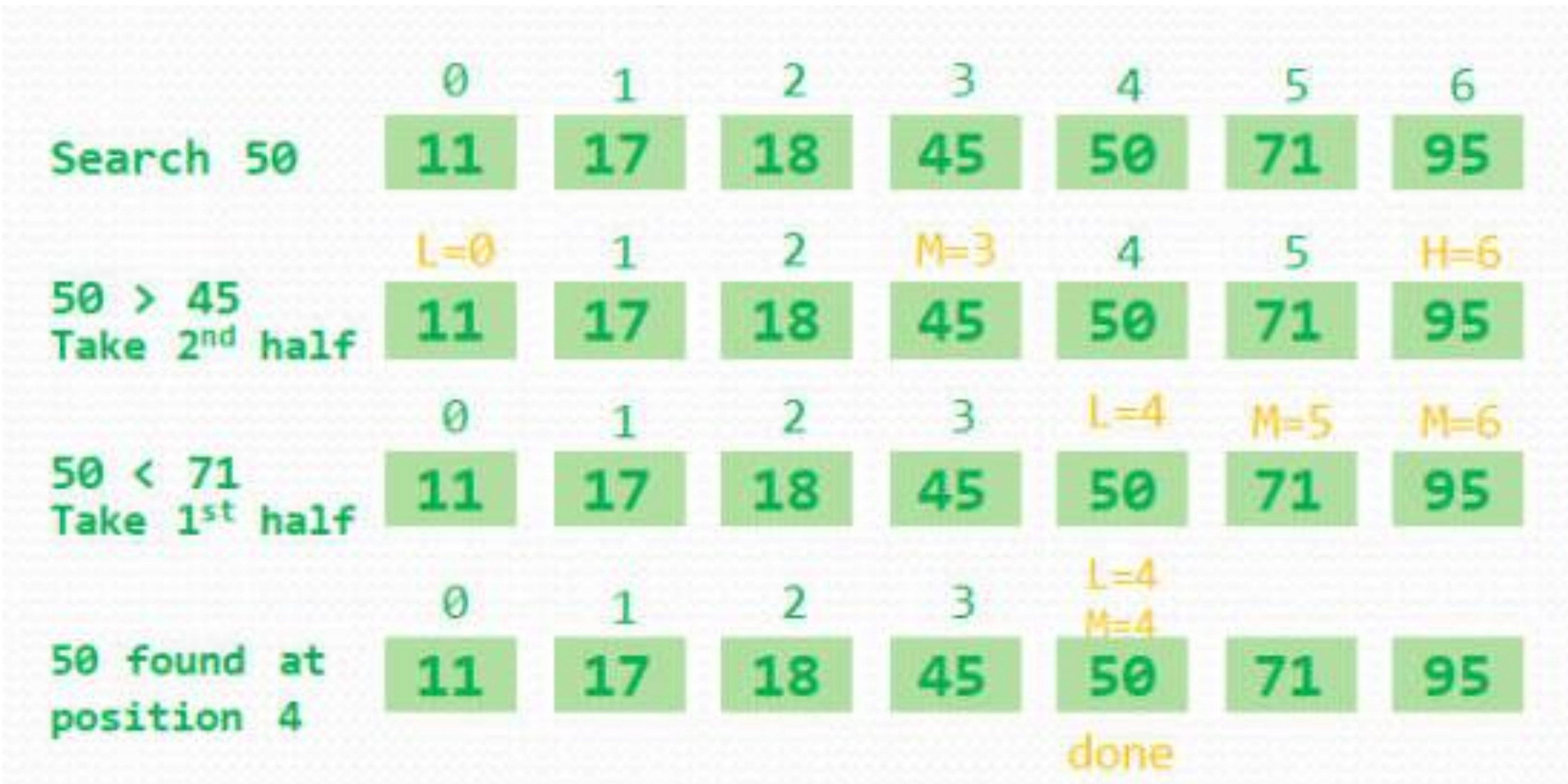
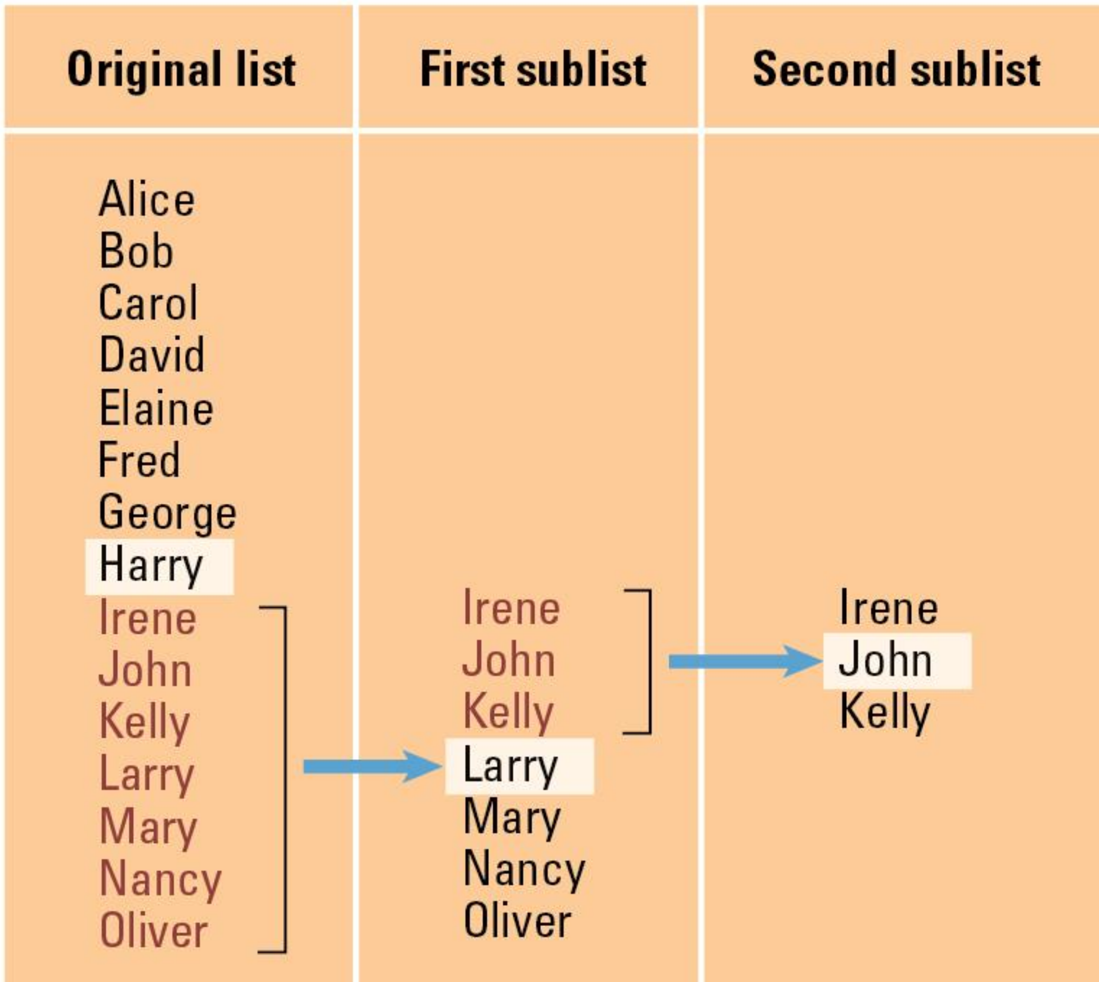


Figure 5.12 Applying our strategy to search a list for the entry John



Pearson

Figure 5.13 A first draft of the binary search technique

```
if (List is empty):
    Report that the search failed
else:
    TestEntry = middle entry in the List
    if (TargetValue == TestEntry):
        Report that the search succeeded
    if (TargetValue < TestEntry):
        Search the portion of List preceding TestEntry for
        TargetValue, and report the result of that search
    if (TargetValue > TestEntry):
        Search the portion of List following TestEntry for
        TargetValue, and report the result of that search
```

Figure 5.14 The binary search algorithm in pseudocode

```
def Search(List, TargetValue):
    if (List is empty):
        Report that the search failed
    else:
        TestEntry = middle entry in the List
        if (TargetValue == TestEntry):
            Report that the search succeeded
        if (TargetValue < TestEntry):
            Sublist = portion of List preceding TestEntry
            Search(Sublist, TargetValue)
        if (TargetValue > TestEntry):
            Sublist = portion of List following TestEntry
            Search(Sublist, TargetValue)
```

Figure 5.15 Recursively Searching

TargetValue = Bill

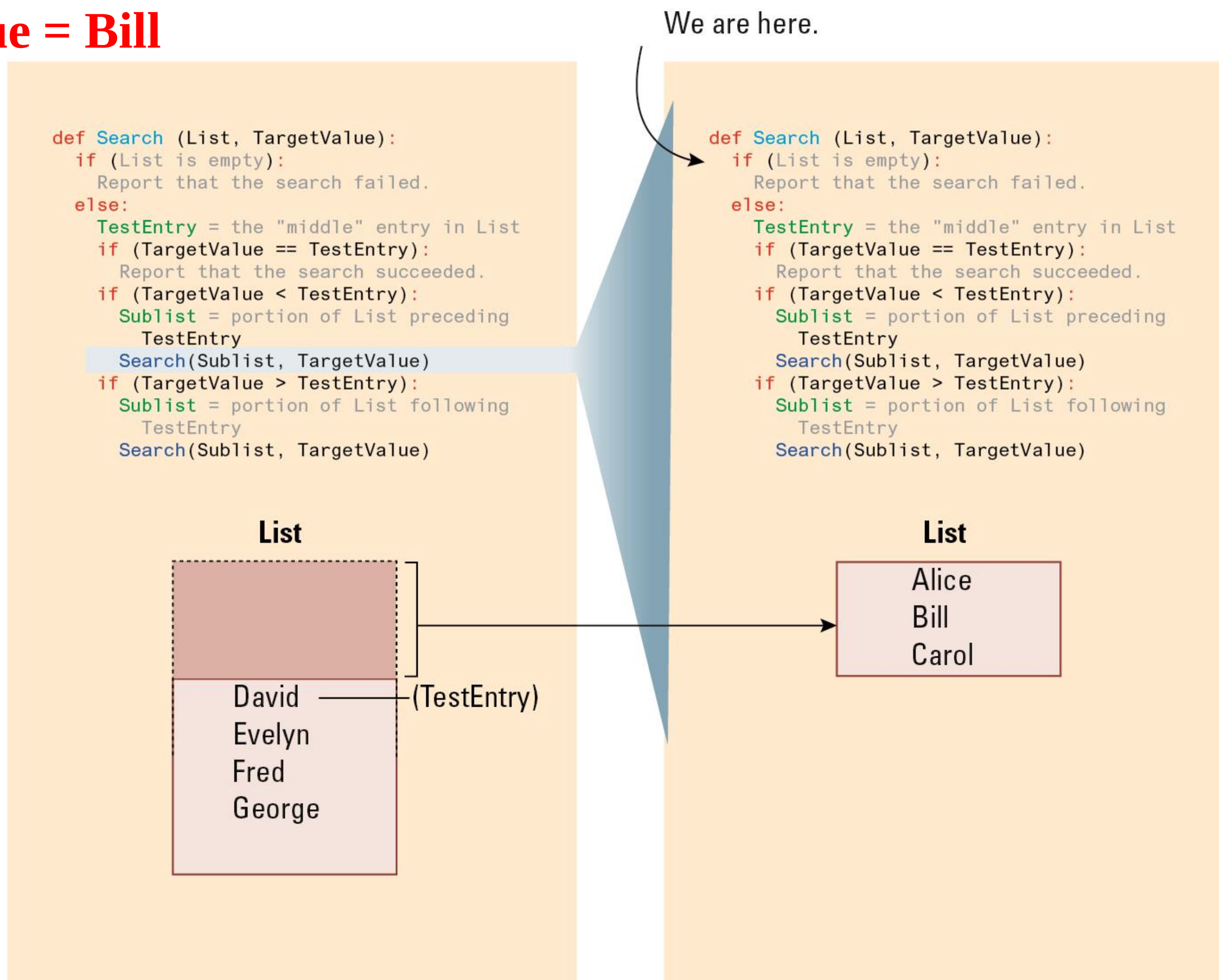


Figure 5.16 Second Recursive Search

TargetValue = David

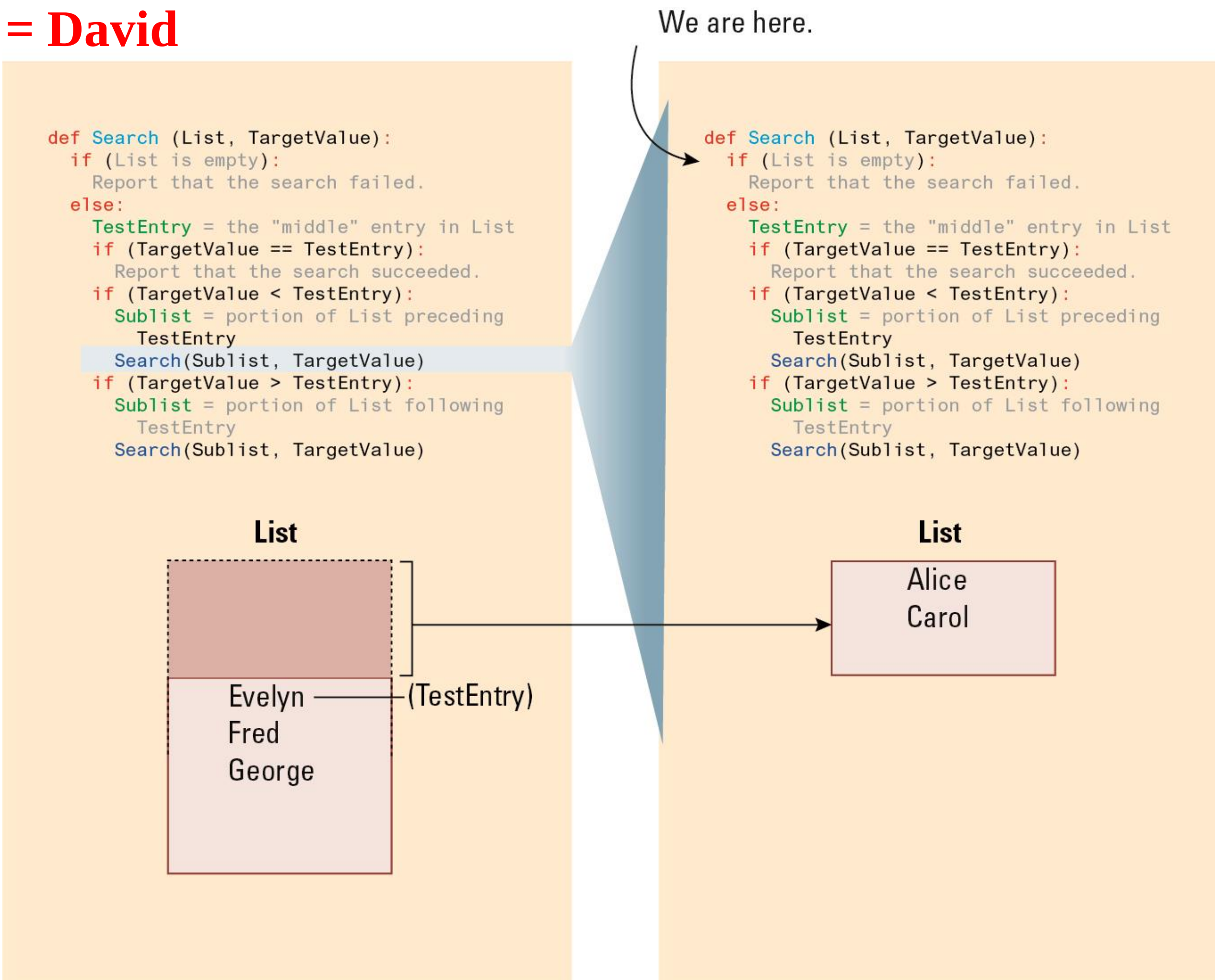
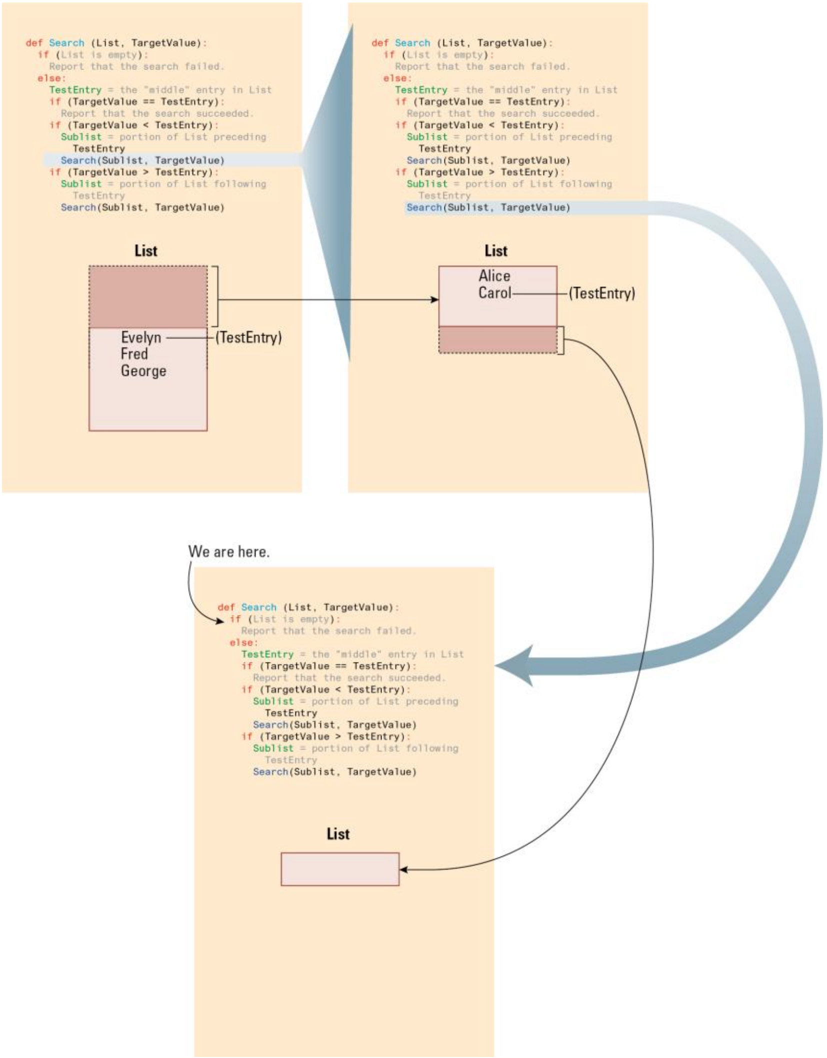


Figure 5.17 Second Recursive Search, Second Snapshot



Recursive Control

- Requires initialization, modification, and a test for termination (base case)
- Provides the illusion of multiple copies of the function, created dynamically in a telescoping manner
- Only one copy is actually running at a given time, the others are waiting

5.6 Efficiency and Correctness

- The choice between efficient and inefficient algorithms can make the difference between a practical solution and an impractical one
- The correctness of an algorithm is determined by reasoning formally about the algorithm, not by testing its implementation

Efficiency

- Measured as number of instructions executed
- Uses big theta notation:
  - Example: Insertion sort is in $\Theta(n^2)$
- Incorporates best, worst, and average case analysis

Figure 5.18 Applying the insertion sort in a worst-case situation

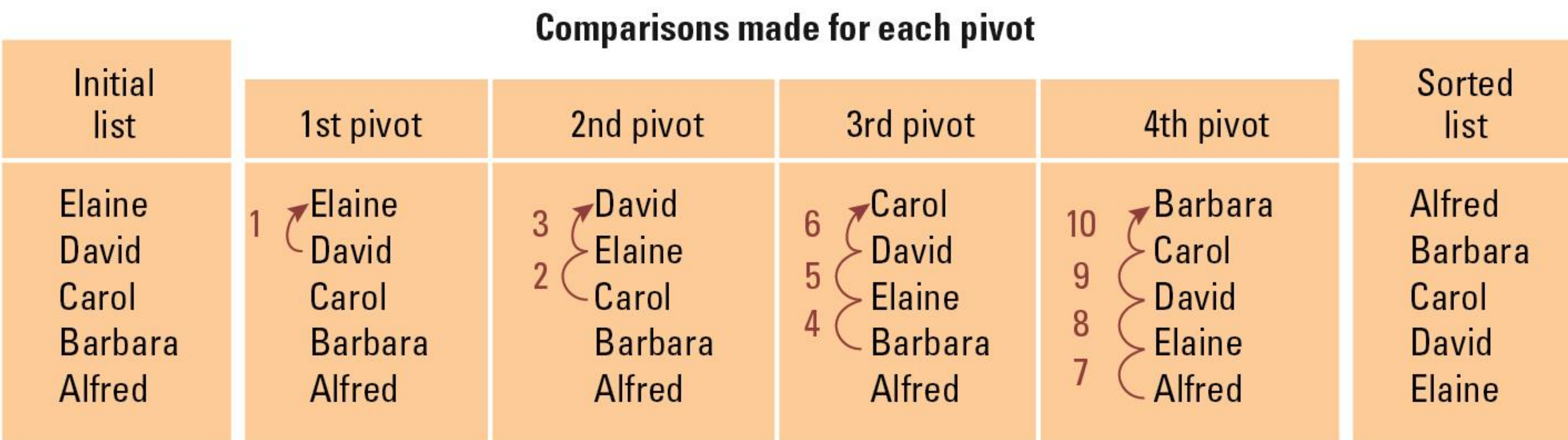


Figure 5.19 Graph of the worst-case analysis of the insertion sort algorithm

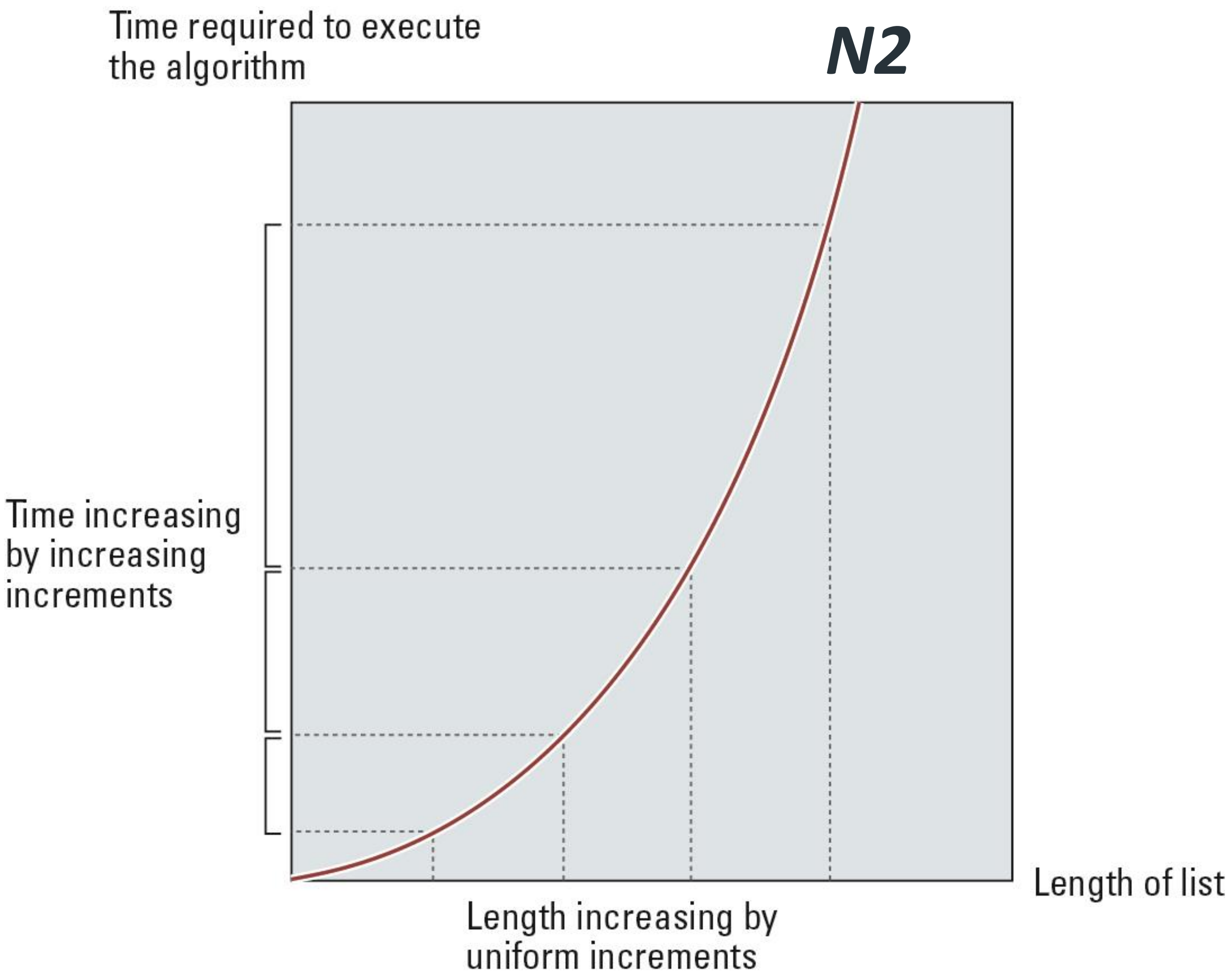
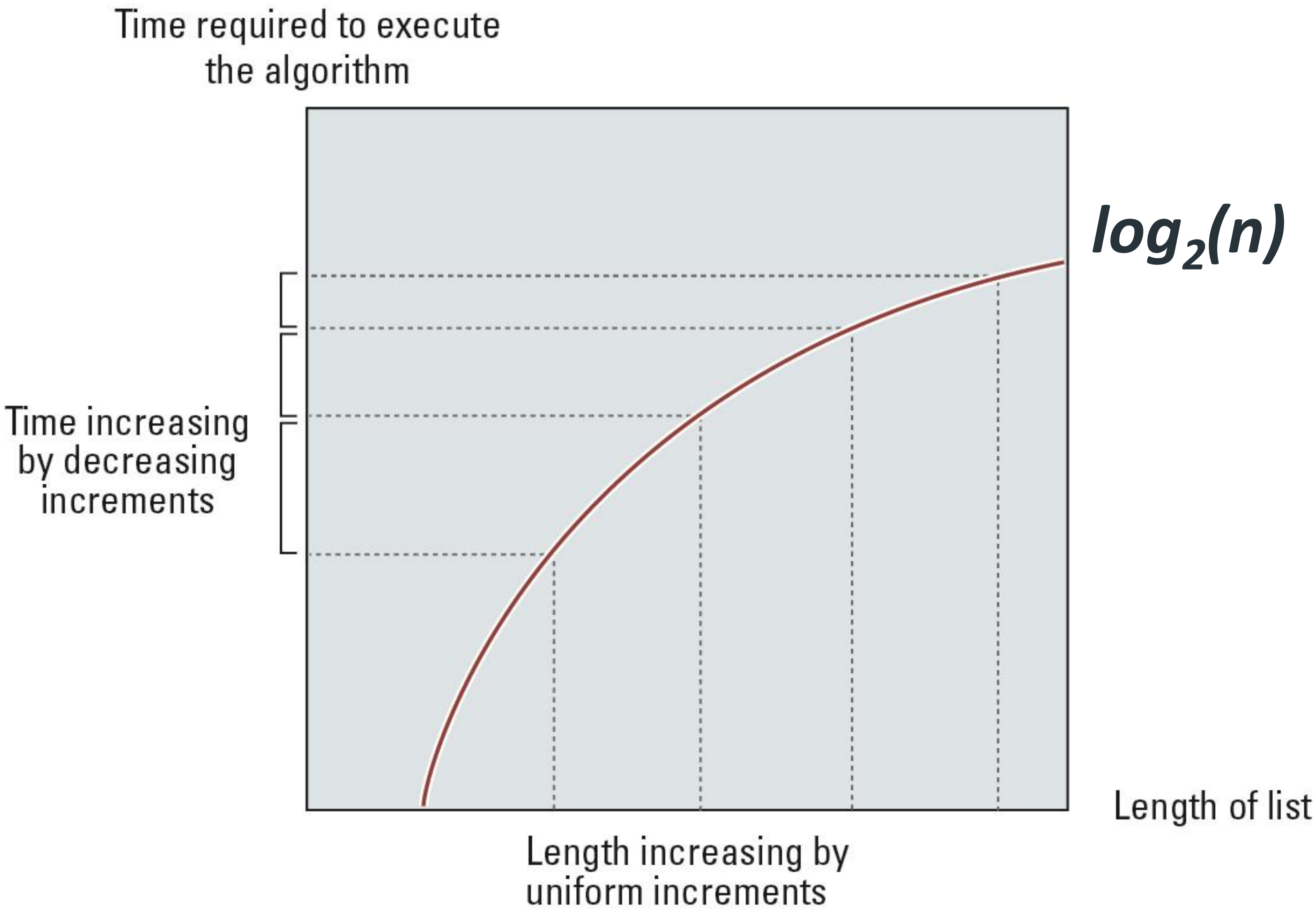


Figure 5.20 Graph of the worst-case analysis of the binary search algorithm



Software Verification

- Proof of correctness (with formal logic)
  - Assertions
    - Preconditions
    - Loop invariants
- Testing is more commonly used to verify software
- Testing only proves that the program is correct for the test cases used

Chain Separating Problem

- A traveler has a gold chain of seven links.
- He must stay at an isolated hotel for seven nights.
- The rent each night consists of one link from the chain.
- What is the fewest number of links that must be cut so that the traveler can pay the hotel one link of the chain each morning without paying for lodging in advance?

Figure 5.21 Separating the chain using only three cuts

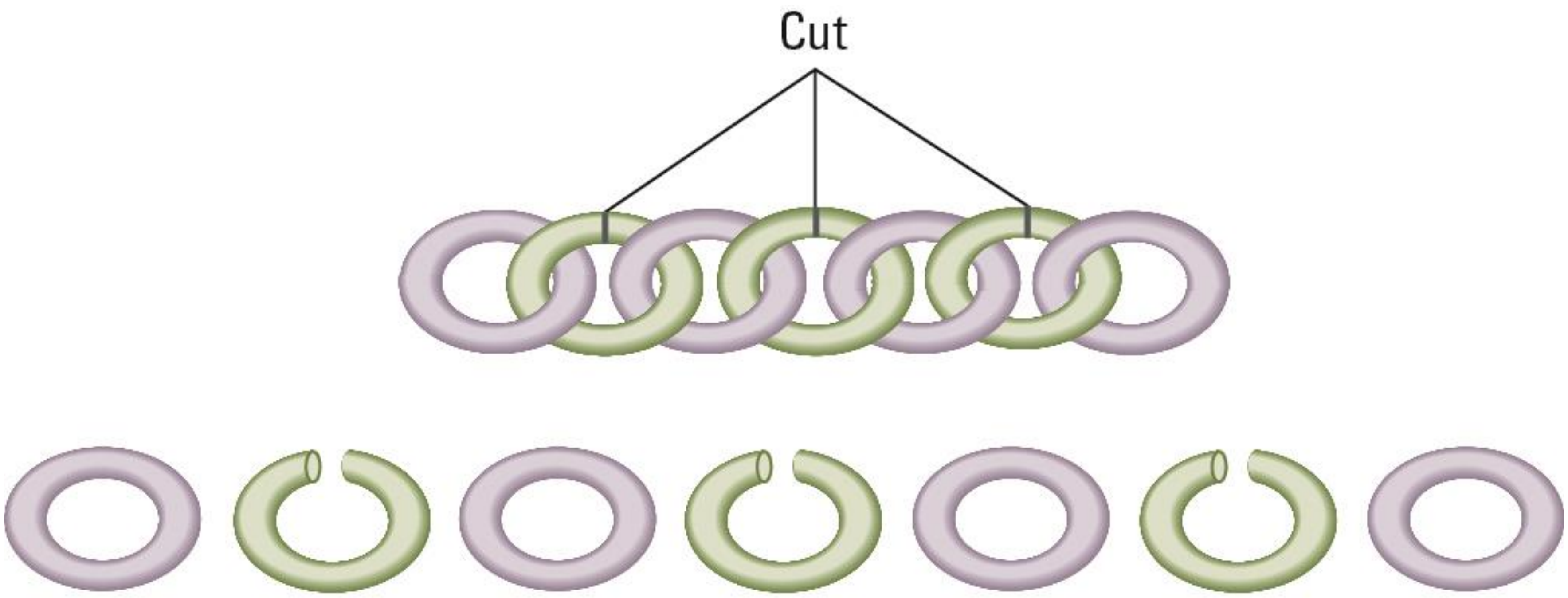


Figure 5.22 Solving the problem with only one cut

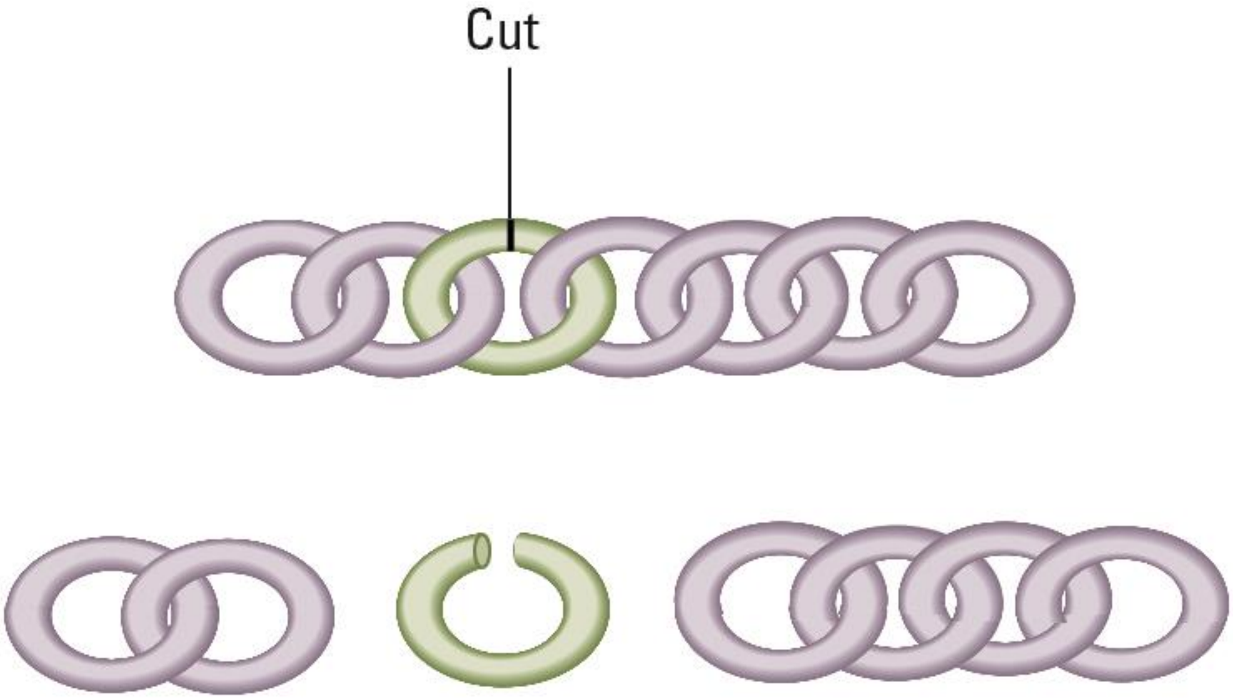


Figure 5.23 The assertions associated with a typical while structure